Local Join Optimization over a Heterogeneously Distributed Scientific Database

Helen X. Xiang
Computer Science, University of Hertfordshire, UK
Email: h.xiang@herts.ac.uk

***Abstract*—This paper begins by discussing the running of the first distributed queries across a heterogeneously distributed scientific database between Manchester and Portsmouth. It then investigates the join operations over the distributed terabyte datasets. Finally, we discuss our experiences of distributed query optimization by pushing the local join operations to the database level.**

I. Introduction

Started in April 2000, the *Sloan Digital Sky Survey* (SDSS) is an influential project building a very detailed digital map of the visible stars and galaxies in the night sky [13], [14]. The SDSS-III is making the map of the Milky Way and searching for extrasolar planets as well as trying to solve the mystery of dark energy [3]. To Date with data release 9 (DR9), the data produced by the survey is summarised in an sixty-terabyte relational database containing photometric objects and spectroscopic information with 14,555 square degrees of the sky coverage. The SDSS data is available to the scientists and the general public via the SkyServer (http://skyserver.sdss.org), the Science Archive Server (SAS) DR9 (http://dr9.sdss3.org) or various mirror sites.

In recent papers we described how we created an experimental distributed version of the SDSS database [6], [7], [9], [11] and experimented with data integration using union queries [12] via the Grid middleware. This is based on OGSA-DQP Distributed Query Processing [15] and the OGSA-DAI middleware[1]. We used OGSA-DAI and OGSA-DQP to integrate the data across different sites—forming a logical distributed database system. Global distributed queries can be processed over this logical database system [8].

In particular, this paper will focus on running local join operations with distributed queries over a distributed SDSS database among different network nodes. We describe the issues when trying to run the local join operation with gigabyte and terabyte datasets.

Please refer to [7], [9] for the details of how we distributed the SDSS MyBestDR5 database system among hosts within the University of Portsmouth and between the universities of Manchester and Portsmouth.

Please refer to [11] for the details of the OGSA-DQP system and its architecture. This paper follows on from our earlier publications [5], [6], [7], [9], [10], [11], [12].

II. Distributing Queries over a Wide Area Network

In the last paper [12], we successfully ran union queries across multiple machines within the University of Portsmouth LAN for the first time. In this paper, we tried to run similar queries on a larger scale.

```
SELECT OBJID
FROM FIRST
WHERE ID>20506
  AND ID<20642
```
(1)

Query 1 uses > and < in the `WHERE` clause to limit the result data to a particular range. It returns 351 rows from the Oracle database `SAND` and 708 rows from SQL Server database `DR5one` on Ace. It returns 1,059 rows when executed against the full SDSS database, taking less than one second.

There was no problem on running the query 1 individually against `SAND` on Manchester Vidar or `DR5one` on Portsmouth Ace through OGSA-DQP. Next we tried the following equivalent OGSA-DQP query on the distributed `BestDR5` database, between Portsmouth Ace and Manchester Vidar over the WAN:

```
(SELECT SAND_FIRST.OBJID FROM SAND_FIRST
  WHERE SAND_FIRST.ID>20506 AND SAND_FIRST.ID<20642)
UNION ALL
(SELECT DR5one_First.objID FROM DR5one_First
  WHERE DR5one_First.id>20506 AND DR5one_First.id<20642)
```
(2)

The OGSA-DQP query 2 failed, but we did not see any obvious error message on the command line output (the query seemed to hang forever).

We enabled the debugging and looked into the DQP evaluator log files for both individually run query 1 and the union query 2. For the failed query 2, running across Ace and Vidar, the log contained this output:

```
Starting next operation on root exchange 2
[...]
Received EOF from input
[...]
service.TransportHandler
(TransportHandler.java:170)
- Exception while sending data:
nested exception is:
java.net.ConnectException: Connection refused
```

The exception message suggests something is blocking the connection for sending the data. We were running the query from Ace in Portsmouth, so we had asked for the Manchester firewall to be configured so evaluator and data service at Manchester were accessible from Portsmouth.

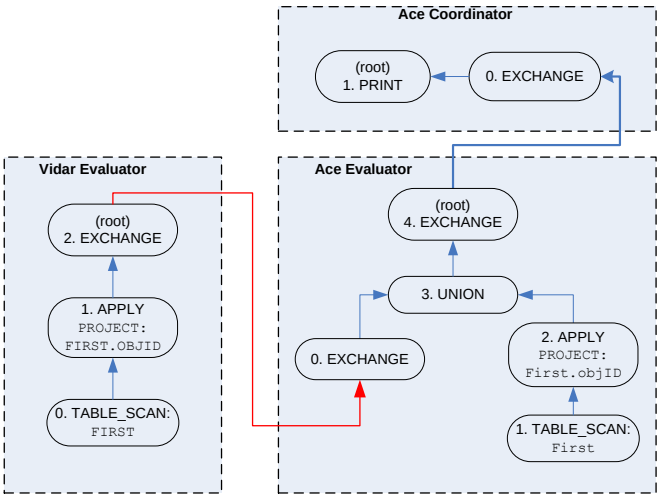

Fig. 1. Query plan for running query 2 on Ace

Figure 1 illustrates the query plan for running query 2 on Ace. The red line in the query plan indicates the data flow from Vidar evaluator to Ace evaluator. This was an unexpected feature—that we have to open the *evaluator on Ace*, because evaluator at Vidar talks back to it. We had only anticipated the need to open up the firewall ports for the evaluator on Vidar, and the coordinator and data service on Ace (but not the firewall port for the evaluator on Ace to receive data from Vidar).

We configured the DQP evaluator on Ace on another port, which is opened through the firewall at the University of Portsmouth. The DQP query 2 ran successfully and returned a $1,059$-row result set in a total time of 5-10 seconds across Ace and Vidar. The OGSA-DQP run-time is related to the memory setting on the DQP Evaluator: more memory leads to less time. For example, it took only five seconds to run query 2 through OGSA-DQP on Ace when its evaluator memory was set to 500MB, but it took about ten seconds if the evaluator memory was below 256MB. (The time for executing the original SDSS query 1 without using OGSA-DQP was less than 1 second.)

We successfully ran union queries across multiple machines between Portsmouth and Manchester, after some firewall problems were resolved.

This was an important milestone in the project—we had run the first distributed query against the full distributed SDSS database successfully. However, this was a simple table-scan type query, involving only one table. We are going to discuss more interesting queries involving joins in the following section.

III. Local Join Optimization

The *join* operation joins two or more tables together, by matching rows from one table to another using one or more column values (*join condition*). It has received considerable attention as one of the most important operations in a relational database [16]. For example, there are many tables and views in the SDSS database, most of them are linked together with some kinds of cross-referencing. For example the `ParentID` of `Galaxy` or `Star` could be cross-referenced to the `objID` in `PhotoObjAll`, and we could find out which stars and galaxies have the same parent object through a join operation like the one in sample query 3 below [8].

```
SELECT Galaxy.ObjID, Galaxy.u, Galaxy.g,
       Galaxy.r, Galaxy.i, Galaxy.z
FROM Galaxy, Star
WHERE Galaxy.parentID > 0
AND Galaxy.parentID = Star.parentID
```
(3)

Optimizing the join operation is important when running the OGSA-DQP queries.

Some of the sample queries in our earlier publications were relatively simple; they only involved one table per query, for example table `PhotoObjAll` in query 1. At the end of an earlier publication [9], we also did some local join queries on a small Oracle database `buck` involving two or three tables. In this paper we are going to further test the join queries involves more tables through OGSA-DQP.

Consider this query:

```
SELECT PHOTOOBJALL.OBJID, PHOTOOBJALL.RA, PHOTOOBJALL.DEC,
       NEIGHBORS.NEIGHBOROBJID, NEIGHBORS.DISTANCE
FROM PHOTOOBJALL, NEIGHBORS
WHERE PHOTOOBJALL.OBJID = NEIGHBORS.OBJID
  AND PHOTOOBJALL.OBJID = 587726015612912497
```
(4)

It returns two rows when executed against the full SDSS database, taking less than 1 second. Under the database partitioning described in the earlier publications [7], [9] , the rows in the result set of this query will come from the SDSS Oracle partition called `buck` in the case of the distributed `MyBestDR5` database (subset SDSS) and the partition called `SAND` in the case of the distributed `BestDR5` database (full SDSS).

We first ran query 4 against `buck` through OGSA-DQP. It took 2 minutes 31 seconds (with debugging turned off). We then ran it against `sand` through OGSA-DQP. In that case the query never finished.

To try to find out what was happening we ran query 4 on `buck` and `sand` through OGSA-DQP with debugging turned on. The size of the coordinator log file on Gizmo (host for `buck`) grew to 568MB while the coordinator log on Vidar (host for `sand`) kept growing larger and larger until we killed the OGSA-DQP query.

The databases `buck` and `sand` have the same SDSS Oracle database schema. For this reason, the query plans on `buck` and `sand` look the same, even though the databases have very different data volumes. Figure 2 shows the query plan for query 4 on the SDSS Oracle databases.

Compared with the first query plan shown in [9], the new feature here is obviously the `HASH_JOIN` operator. The `HASH_JOIN` operator takes the outputs from `APPLY` operations 1 and 3 and joins them together under the condition that `PHOTOOBJALL.OBJID` is equal to `NEIGHBORS.OBJID`. The joined result data is then projected in the `APPLY` operation 5 to give the five requested columns, before being sent to the root evaluator in the coordinator.

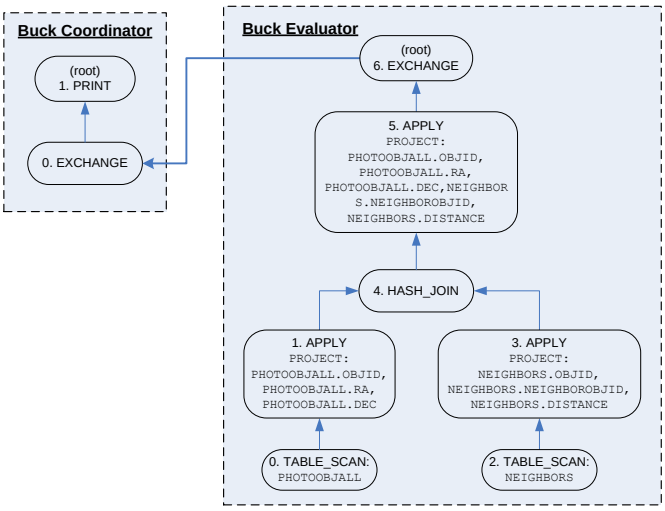

Fig. 2. Query plan for query 4 on the SDSS Oracle database

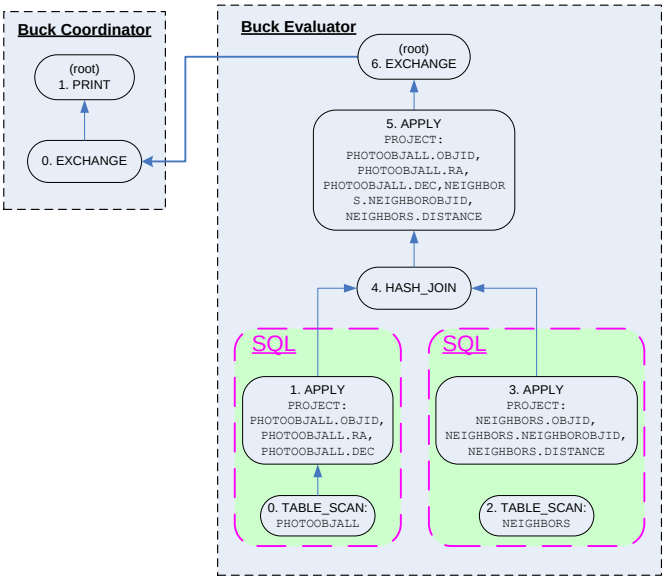

Fig. 3. Query plan for query 4 on `buck` highlighting two SQL Queries

After applying our optimization described in [9] the `TABLE_SCAN/APPLY` sub-trees of the query plan form two SQL queries directly executed against the database. The green blobs in the query plan of Figure 3 highlight these two SQL queries. The `HASH_JOIN/APPLY` sub-tree are computed from the result sets within the evaluator itself.

Looking at the source code for the OGSA-DQP evaluator, we find `ObjectBuilder` converts the `HASH_JOIN` node to a `HashJoinOp` object. The documentation comment on the `Open()` method of this class says:

> The `Open()` method opens the left input, recursively calls the next method for the left input and hashes each tuple to the hash table, until the left input is exhausted. Then it opens the right input.

The documentation comment on the `Next()` method says:

> The `Next()` method fetches the next tuples from the right input, applies the same hash function and probes the hash table for matches. If a match is found, the two tuples are joined and verified against the predicate. If the predicate evaluation returns true, the tuple is returned to the upper operator, otherwise the next tuple is fetched from the right input—this process is repeated till the `EOF` is found.

By *the tuple is returned to the upper operator*, they just mean that the `Next()` method returns this tuple to the operator above in the query plan.

What this is saying is that the whole of the left input is loaded into evaluator memory. The right input is streamed in and selected or discarded row by row. In our example the left input is only going to be a single selected row from `PHOTOOBJALL`. But all rows from `NEIGHBORS` will be streamed through the evaluator as the right input. That is a table with a *lot* of rows .

To find out exactly how many rows the `HASH_JOIN` operation of query 4 has to deal with, we need to take a look at the SQL queries in the `TABLE_SCAN/APPLY` sub-trees of the Figure 3. The SQL query for `TABLE_SCAN(0)/APPLY(1)` sub-tree is:

```
SELECT PHOTOOBJALL.OBJID, PHOTOOBJALL.RA, PHOTOOBJALL.DEC
FROM PHOTOOBJALL
WHERE PHOTOOBJALL.OBJID = 587726015612912497
```
(5)

The SQL query for `TABLE_SCAN(2)/APPLY(3)` sub-tree is:

```
SELECT NEIGHBORS.NEIGHBOROBJID, NEIGHBORS.DISTANCE
FROM NEIGHBORS
```
(6)

SQL query 5 returns only 1 row on either `buck` or `SAND`. But SQL query 6 returns $355,214$ rows from the `buck` database and $1,431,788,773$ rows from the `SAND` database. The evaluator has to inspect all these intermediate result data to find the two rows in the final result. Processing all these intermediate result data in the evaluator is incredibly slowing down the whole query process.

Having to process large numbers of intermediate data in the evaluator is the reason why the query 4 through OGSA-DQP is relatively slow on `buck` and appears to hang indefinitely on `sand`. This also explained why the evaluator log grew so large during the OGSA-DQP query processing with debugging enabled.

Clearly what we would like is for the local join to be done in SQL. Figure 4 captures such an optimization, abstracting all the `TABLE_SCAN`, `APPLY` and `HASH_JOIN` operations into a single SQL query.

So we want to do some more optimization on the evaluator, to amalgamate local joins into single SQL queries that can be executed at the database level, therefore avoiding the intermediate data flooding the evaluator. We now sketch an optimization to local joins that involves intervention in classes `TableScanOp`, `Operator`, `BaseOperator`, `ReduceOp`, and `JoinAndEvaluateBaseOperator`, and introduces a new class called `SQLSelect`.

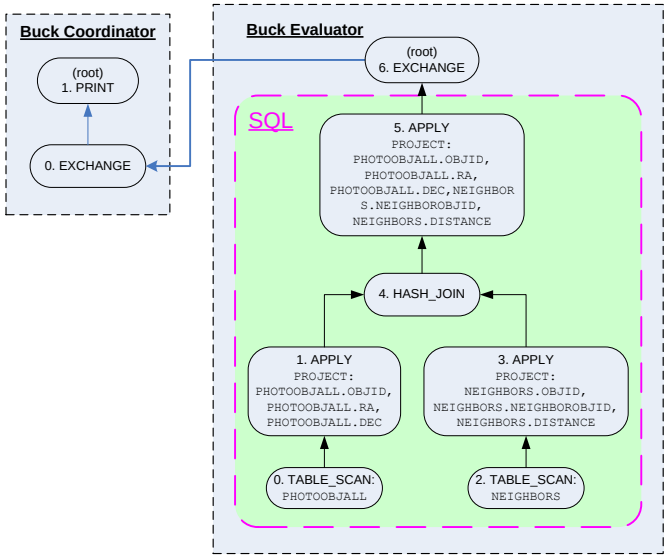

Fig. 4. Query plan for query 4 on `buck` highlighting single SQL Query after optimization.

In the original version of the OGSA-DQP evaluator, direct SQL queries against the local database are usually made by `TableScanOp`[1]. This operator executes a query of the general form:

`SELECT * FROM` *table* `WHERE` <*predicate*>

As we have explained, this is the most complex kind of query that OGSA-DQP normally executes against the underlying SQL databases. More complex queries are built from these queries by processing their output, within the evaluator.

We implemented our optimization code by first generalizing this original `TableScanOp`, and renaming it to `SQLSelect`. The new class can execute a query of the general form:

`SELECT` <*select-list*> `FROM` *table1*, ..., *tableN* `WHERE` <*predicate*>

The constructor for this class takes a list of tables, a "tuple type" that includes the list of fields to select, and the predicate. We added two mutator methods called `project()` and `join()`. The first mutator changes the SQL query created by the constructor, replacing the tuple type with one that projects the selection list. The second mutator takes a second `SQLSelect` and a join condition (and a tuple type). The tables in the `SQLSelect` argument are added to the tables of the original query, and the predicates are "*and*ed" together with the join condition. Like all OGSA-DQP operators, our `SQLSelect` class follows the *iterator* model [17] and implements methods `Open()`, `Next()` and `Close()` to run the query, then iterate through its result set.

Now the base class of the OGSA-DQP operators is modified to add a property `sqlOp` of type `SQLSelect`.

[1]The `HashLoopJoinOp` can also make direct queries, but we did not usually see this in our query plans.

The `TableScanOp` class now becomes very simple. Its constructor just initializes the `sqlOp` property with a `SQLSelect` containing a single table and a tuple type equivalent to selecting "`*`". Its `Open()`, `Next()` and `Close()` methods just call the same methods on `sqlOp`.

The OGSA-DAI `ReduceOp` is characterized by an input operator `inputOp` and an operation type `applyOpType`. We modified this class so that its `Open()` method does something like[2]:

```
SQLSelect inputSql = inputOp.getSqlOp() ;
if(inputSql != null && "PROJECT".equals(applyOpType)) {
  sqlOp = inputSql.project(tupleType) ;
}
if(sqlOp != null) {
  return sqlOp.Open();
}
... else use original implementation of Open()...
```

In other words, if the input operation has a `SQLSelect` equivalent, and the reduction is a simple projection operation (the commonest kind of reduction), the input `SQLSelect` is projected and the result becomes the `SQLSelect` replacement for this class, which is opened. Otherwise the original implementation is used. The `Next()` method looks something like this:

```
if(sqlOp != null) {
  return sqlOp.Next();
}
... else use original implementation of Next()...
```

A join operator is characterised by a left input, a right input, and a join condition. We modified the `Open()` method to something like:

```
SQLSelect leftSql  = leftInput.getSqlOp() ;
SQLSelect rightSql = rightInput.getSqlOp() ;
if(leftSql != null && rightSql != null &&
    ... input queries refer to same local database ...) {
  sqlOp = leftSql.join(rightSql, mExpression, tupleType) ;
}
if(sqlOp != null) {
  return sqlOp.Open();
}
... else use original implementation of Open()...
```

So if the input operations both have a `SQLSelect` equivalent referring to the same database, these `SQLSelect`s are joined and the result becomes the `SQLSelect` replacement for this class, which is opened. Otherwise the original implementation is used.

This recursive approach ensures that if all child components of the query have a "SQL equivalent", the whole query is implemented as local SQL. If (for example) any child is an exchange operator, there will be no SQL equivalent, and the original implementation is used.

After implementing these changes to the OGSA-DQP, we ran query 4 through OGSA-DQP against both `buck` and `SAND`. The OGSA-DQP query on `buck` now took only 12 seconds with debugging turned off. Compared with the original time of 2 minutes 31 seconds before the optimization, this is more then 12 times faster. More importantly, the query on `SAND` also finished within seconds (it never finished at all before the optimization). The coordinator log on Vidar is also

[2]This is pseudo code. The real logic is a bit more complicated.

relatively small. (The time for executing the original SDSS query 4 without using OGSA-DQP was less than 1 second.)

As well as running the test query on the Oracle databases, we also ran a similar query against the `DR5one` SQL Server database through the optimized OGSA-DQP and got the correct result back within seconds.

Summary and Future Work

In this paper we first showed how to run a query against a distributed version of the full SDSS database using OGSA-DQP. We then identified a problem when running the local join operation through OGSA-DQP. With the standard implementation of OGSA-DQP, the local join queries ran on a gigabyte database (slowly) but failed to run on the terabyte database. We rewired the OGSA-DQP evaluator and pushed the local join operation to the database level. Our solution produced a dramatic improvement to the query performance. The local join queries now run much faster in both gigabyte and terabyte databases. In future publications we will investigate distributed join queries, involving *cross-joins*.

References

[1] The OGSA-DAI project home page. http://www.ogsdai.org.

[2] The OGSA-DQP project. http://www.ogsadai.org/about/ogsa-dqp.

[3] SDSS-III http://http://www.sdss3.org/.

[4] Sun Microsystems, Inc. Mapping SQL and Java types. http://java.sun.com/j2se/1.5.0/docs/guide/jdbc/getstart/mapping.html.

[5] H. Xiang, M. Baker, and R. Nichol. Experiences mirroring and distributing the Sloan Digital Sky Survey. In *Fifth International Conference on Grid and Cooperative Computing Workshops (GCC 2006), Changsha, China*, pages 518–521. IEEE Computer Society, October 2006.

[6] H. X. Xiang. Experiences acquiring and distributing a large scientific database. *Future Generation Communication and Networking Symposia*, 2:14–19, 2008.

[7] H. X. Xiang. A grid-based distributed database solution for large astronomy datasets. *Computer Science and Software Engineering*, 3:66–69, 2008.

[8] H. X. Xiang. *A Grid-based Distributed Database Solution for Large Astronomy Datasets*. PhD thesis, Portsmouth, UK, February 2008.

[9] H. X. Xiang. Experiences running ogsa-dqp queries against a heterogeneous distributed scientific database. In *ICPADS '09: Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems*, pages 706–710, Washington, DC, USA, 2009. IEEE Computer Society.

[10] H. X. Xiang. Supporting complex scientific database schemas in a grid middleware. In *AINA '09: Proceedings of the 2009 International Conference on Advanced Information Networking and Applications*, pages 937–944, University of Bradford, Bradford, UK, May 2009. IEEE Computer Society.

[11] H. X. Xiang. Using grid middleware to query a heterogeneous distributed version of the sdss database. In *ICPADS '09: Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems*, pages 870–875, Washington, DC, USA, 2009. IEEE Computer Society.

[12] H. X. Xiang. Integrated Queries over a Heterogeneously Distributed Scientific Database using OGSA-DQP In *ITAIC 2011: Proceedings of the 2011 6th IEEE Joint International Information Technology and Artificial Intelligence Conference*, 421-425, Washington, DC, USA, 2009. IEEE Computer Society.

[13] D. G. York et al. The Sloan Digital Sky Survey: Technical summary. *Astronomical Journal*, 120:1579–1587, 2000. http://www.sdss.org.

[14] H. Aihara et al. The The Eighth Data Release of the Sloan Digital Sky Survey: First Data from SDSS-III. *The Astrophysical Journal Supplement Series*, 193:29, 2011. http://stacks.iop.org/0067-0049/193/i=2/a=29

[15] S. Lynden, A. Mukherjee, A. C. Hume,A. A. A. Fernandes, N. W. Paton, R. Sakellariou, and P. Watson The design and implementation of OGSA-DQP: A service-based distributed query processor. In *Future Generation Computer Systems*, 25: 224-236, March 2009, Elsevier. http://www.cs.man.ac.uk/ norm/publications.php

[16] L. D. Shapiro Join processing in database systems with large main memories. In *ACM Trans. Database Syst.*, 11: 2239–264, New York, NY, USA, 1986. ACM Press. http://www.cs.man.ac.uk/ norm/publications.php

[17] G. Graefe Query evaluation techniques for large databases. In *ACM Computing Surveys*, 25: 73-170, June 1993. ACM Press.